

# F (FNOP — FYL2XP1)

45

## 45.1 FNOP—No Operation

Opcode	Instruction	Description
D9 D0	FNOP	No operation is performed.

### Description

Performs no FPU operation. This instruction takes up space in the instruction stream but does not affect the FPU or machine context, except the EIP register.

### FPU Flags Affected

C0, C1, C2, C3 undefined.

### Floating-Point Exceptions

None.

### Protected Mode Exceptions

#NM EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#NM EM or TS in CR0 is set.

## 45.2 FPATAN—Partial Arctangent

Opcode	Instruction	Description
D9 F3	FPATAN	Replace ST(1) with $\arctan(\text{ST}(1)/\text{ST}(0))$ and pop the register stack

### Description

Computes the arctangent of the source operand in register ST(1) divided by the source operand in register ST(0), stores the result in ST(1), and pops the FPU register stack. The result in register ST(0) has the same sign as the source operand ST(1) and a magnitude less than  $+\pi$ .

The FPATAN instruction returns the angle between the X axis and the line from the origin to the point (X,Y), where Y (the ordinate) is ST(1) and X (the abscissa) is ST(0). The angle depends on the sign of X and Y independently, not just on the sign of the ratio Y/X. This is because a point (–

X,Y) is in the second quadrant, resulting in an angle between  $\pi/2$  and  $\pi$ , while a point (X,-Y) is in the fourth quadrant, resulting in an angle between 0 and  $-\pi/2$ . A point (-X,-Y) is in the third quadrant, giving an angle between  $-\pi/2$  and  $-\pi$ .

The following table shows the results obtained when computing the arctangent of various classes of numbers, assuming that underflow does not occur.

		ST(0)						
ST(1)		-•	-F	-0	+0	+F	+∞	NaN
	-•	-3π/4*	-π/2	-π/2	-π/2	-π/2	-π/4*	NaN
	-F	-p	-π to -π/2	-π/2	-π/2	-π/2 to -0	-0	NaN
	-0	-p	-p	-p*	-0*	-0	-0	NaN
	+0	+π	+π	+π*	+0*	+0	+0	NaN
	+F	+π	+π to +π/2	+π/2	+π/2	+π/2 to +0	+0	NaN
	+∞	+3π/4*	+π/2	+π/2	+π/2	+π/2	+π/4*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

#### NOTES:

1. FMeans finite-real number.
2. \*Table "Invalid Arithmetic Operations and the Masked Responses to Them", specifies that the ratios  $0/0$  and  $\infty/\infty$  generate the floating-point invalid arithmetic-operation exception and, if this exception is masked, the real indefinite value is returned. With the FPATAN instruction, the  $0/0$  or  $\infty/\infty$  value is actually not calculated using division. Instead, the arctangent of the two variables is derived from a standard mathematical formulation that is generalized to allow complex numbers as arguments. In this complex variable formulation,  $\arctangent(0,0)$  etc. has well defined values. These values are needed to develop a library to compute transcendental functions with complex arguments, based on the FPU functions that only allow real numbers as arguments.

There is no restriction on the range of source operands that FPATAN can accept.

## Intel Architecture Compatibility

The source operands for this instruction are restricted for the 80287 math coprocessor to the following range:

$$0 \leq |ST(1)| < |ST(0)| < +\infty$$

## Operation

```
ST(1) ← arctan(ST(1) / ST(0));
PopRegisterStack;
```

## FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
- Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
- C0, C2, C3 Undefined.

## Floating-Point Exceptions

- #IS Stack underflow occurred.

#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## 45.3 FPATAN—Partial Arctangent

Opcode	Instruction	Description
D9 F8	FPREM	Replace ST(0) with the remainder obtained from dividing ST(0) by ST(1)

### Description

Computes the remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

$$\text{Remainder} = \text{ST}(0) - (Q * \text{ST}(1))$$

Here, Q is an integer value that is obtained by truncating the real-number quotient of [ST(0) / ST(1)] toward zero. The sign of the remainder is the same as the sign of the dividend. The magnitude of the remainder is less than that of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

		ST(1)					
		-∞	-F	-0	+0	+F	+∞
ST(0)	-∞	*	*	*	*	*	*
	-F	ST(0)	-F or -0	**	**	-F or -0	ST(0)
	-0	-0	-0	*	*	-0	-0
	+0	+0	+0	*	*	+0	+0
		NaN	NaN	NaN	NaN	NaN	NaN

+F	ST(0)	+F or +0	**	**	+F or +0	ST(0)	NaN
$+\infty$	*	*	*	*	*	*	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**NOTES:**

1. FMeans finite-real number.
2. \*Indicates floating-point invalid-arithmetic-operand (#IA) exception.
3. \*\*Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is  $\infty$ , the result is equal to the value in ST(0).

The FPREM instruction does not compute the remainder specified in IEEE Std 754. The IEEE specified remainder can be computed with the FPREM1 instruction. The FPREM instruction is provided for compatibility with the Intel 8087 and Intel287 math coprocessors.

The FPREM instruction gets its name “partial remainder” because of the way it computes the remainder. This instructions arrives at a remainder through iterative subtraction. It can, however, reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU status word. This information is important in argument reduction for the tangent function (using a modulus of  $\pi/4$ ), because it locates the original angle in the correct one of eight sectors of the unit circle.

## Operation

```

D ← exponent(ST(0)) - exponent(ST(1));
IF D < 64
    THEN
        Q ← Integer(TruncateTowardZero(ST(0) / ST(1)));
        ST(0) ← ST(0) - (ST(1) * Q);
        C2 ← 0;
        C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 ← 1;
        N ← an implementation-dependent number between 32 and 63;
        QQ ← Integer(TruncateTowardZero((ST(0) / ST(1)) / 2(D-N)));
        ST(0) ← ST(0) - (ST(1) * QQ * 2(D-N));
FI;

```

## FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

## Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus is 0, dividend is $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

## Protected Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

# 45.4 FPREM1—Partial Remainder

Opcode	Instruction	Description
D9 F5	FPREM1	Replace ST(0) with the IEEE remainder obtained from dividing ST(0) by ST(1)

## Description

Computes the IEEE remainder obtained from dividing the value in the ST(0) register (the dividend) by the value in the ST(1) register (the divisor or **modulus**), and stores the result in ST(0). The remainder represents the following value:

$$\text{Remainder} = \text{ST}(0) - (Q * \text{ST}(1))$$

Here, Q is an integer value that is obtained by rounding the real-number quotient of  $[\text{ST}(0) / \text{ST}(1)]$  toward the nearest integer value. The magnitude of the remainder is less than half the magnitude of the modulus, unless a partial remainder was computed (as described below).

This instruction produces an exact result; the precision (inexact) exception does not occur and the rounding control has no effect. The following table shows the results obtained when computing the remainder of various classes of numbers, assuming that underflow does not occur.

		ST(1)					
		-∞	-F	-0	+0	+F	+∞
		-∞	*	*	*	*	*
ST(0)	-F	ST(0)	$\pm F$ or -0	**	**	$\pm F$ or -0	ST(0)
	-0	-0	-0	*	*	-0	-0
		NaN					

+0	+0	+0	*	*	+0	+0	NaN
+F	ST(0)	±F or +0	**	**	±F or +0	ST(0)	NaN
+∞	*	*	*	*	*	*	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**NOTES:**

1. FMeans finite-real number.
2. \*Indicates floating-point invalid-arithmetic-operand (#IA) exception.
3. \*\*Indicates floating-point zero-divide (#Z) exception.

When the result is 0, its sign is the same as that of the dividend. When the modulus is  $\infty$ , the result is equal to the value in ST(0).

The FPREM1 instruction computes the remainder specified in IEEE Std 754. This instruction operates differently from the FPREM instruction in the way that it rounds the quotient of ST(0) divided by ST(1) to an integer (see the “Operation” section below).

Like the FPREM instruction, the FPREM1 computes the remainder through iterative subtraction, but can reduce the exponent of ST(0) by no more than 63 in one execution of the instruction. If the instruction succeeds in producing a remainder that is less than one half the modulus, the operation is complete and the C2 flag in the FPU status word is cleared. Otherwise, C2 is set, and the result in ST(0) is called the **partial remainder**. The exponent of the partial remainder will be less than the exponent of the original dividend by at least 32. Software can re-execute the instruction (using the partial remainder in ST(0) as the dividend) until C2 is cleared. (Note that while executing such a remainder-computation loop, a higher-priority interrupting routine that needs the FPU can force a context switch in-between the instructions in the loop.)

An important use of the FPREM1 instruction is to reduce the arguments of periodic functions. When reduction is complete, the instruction stores the three least-significant bits of the quotient in the C3, C1, and C0 flags of the FPU status word. This information is important in argument reduction for the tangent function (using a modulus of  $\pi/4$ ), because it locates the original angle in the correct one of eight sectors of the unit circle.

## Operation

```

D ← exponent(ST(0)) - exponent(ST(1));
IF D < 64
    THEN
        Q ← Integer(RoundTowardNearestInteger(ST(0) / ST(1)));
        ST(0) ← ST(0) - (ST(1) * Q);
        C2 ← 0;
        C0, C3, C1 ← LeastSignificantBits(Q); (* Q2, Q1, Q0 *)
    ELSE
        C2 ← 1;
        N ← an implementation-dependent number between 32 and 63;
        QQ ← Integer(TruncateTowardZero((ST(0) / ST(1)) / 2(D-N)));
        ST(0) ← ST(0) - (ST(1) * QQ * 2(D-N));
FI;
```

## FPU Flags Affected

C0	Set to bit 2 (Q2) of the quotient.
C1	Set to 0 if stack underflow occurred; otherwise, set to least significant bit of quotient (Q0).
C2	Set to 0 if reduction complete; set to 1 if incomplete.
C3	Set to bit 1 (Q1) of the quotient.

## Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, modulus (divisor) is 0, dividend is $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.

## Protected Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

# 45.5 FPTAN—Partial Tangent

Opcode	Instruction	Clocks	Description
D9 F2	FPTAN	17-173	Replace ST(0) with its tangent and push 1 onto the FPU stack.

## Description

Computes the tangent of the source operand in register ST(0), stores the result in ST(0), and pushes a 1.0 onto the FPU register stack. The source operand must be given in radians and must be less than  $\pm 2^{63}$ . The following table shows the unmasked results obtained when computing the partial tangent of various classes of numbers, assuming that underflow does not occur.

ST(0) SRC	ST(0) DEST
–•	*
–F	–F to +F
–0	–0
+0	+0
+F	–F to +F
$+\infty$	*
NaN	NaN

### NOTES:

1. FMeans finite-real number.
2. \*Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range  $-2^{63}$  to  $+2^{63}$  can be reduced to the range of the instruction by subtracting an appropriate integer multiple of  $2\pi$  or by using the FPREM instruction with a divisor of  $2\pi$ . See “Pi”, for a discussion of the proper value to use for  $\pi$  in performing such reductions.

The value 1.0 is pushed onto the register stack after the tangent has been computed to maintain compatibility with the Intel 8087 and Intel287 math coprocessors. This operation also simplifies the calculation of other trigonometric functions. For instance, the cotangent (which is the reciprocal of the tangent) can be computed by executing a FDIVR instruction after the FPTAN instruction.

## Operation

```
IF ST(0) < 263
THEN
    C2 ← 0;
    ST(0) ← tan(ST(0));
    TOP ← TOP - 1;
    ST(0) ← 1.0;
ELSE (*source operand is out-of-range *)
    C2 ← 1;
FI;
```

## FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C2	Set to 1 if source operand is outside the range $-2^{63}$ to $+2^{63}$ ; otherwise, cleared to 0.
C0, C3	Undefined.

## Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#P	Value cannot be represented exactly in destination format.

## Protected Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## 45.6 FRNDINT—Round to Integer

Opcode	Instruction	Description
D9 FC	FRNDINT	Round ST(0) to an integer.

### Description

Rounds the source value in the ST(0) register to the nearest integral value, depending on the current rounding mode (setting of the RC field of the FPU control word), and stores the result in ST(0).

If the source value is  $\infty$ , the value is not changed. If the source value is not an integral value, the floating-point inexact-result exception (#P) is generated.

### Operation

```
ST(0) ← RoundToIntegerValue(ST(0));
```

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#P	Source operand is not an integral value.

### Protected Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## 45.7 FRSTOR—Restore FPU State

Opcode	Instruction	Description
DD /4	FRSTOR <i>m94/108byte</i>	Load FPU state from <i>m94byte</i> or <i>m108byte</i> .

### Description

Loads the FPU state (operating environment and register stack) from the memory area specified with the source operand. This state data is typically written to the specified memory location by a previous FSAVE/FNSAVE instruction.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures “Protected Mode FPU State Image in Memory, 32-Bit Format” through “Real Mode FPU State Image in Memory, 16-Bit Format”, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The FRSTOR instruction should be executed in the same operating mode as the corresponding FSAVE/FNSAVE instruction.

If one or more unmasked exception bits are set in the new FPU status word, a floating-point exception will be generated. To avoid raising exceptions when loading a new operating environment, clear all the exception flags in the FPU status word that is being loaded.

### Operation

```

FPUControlWord ← SRC(FPUControlWord);
FPUStatusWord ← SRC(FPUStatusWord);
FPUTagWord ← SRC(FPUTagWord);
FPUDataPointer ← SRC(FPUDataPointer);
FPUInstructionPointer ← SRC(FPUInstructionPointer);
FPULastInstructionOpcode ← SRC(FPULastInstructionOpcode);
ST(0) ← SRC(ST(0));
ST(1) ← SRC(ST(1));
ST(2) ← SRC(ST(2));
ST(3) ← SRC(ST(3));
ST(4) ← SRC(ST(4));
ST(5) ← SRC(ST(5));
ST(6) ← SRC(ST(6));
ST(7) ← SRC(ST(7));

```

### FPU Flags Affected

The C0, C1, C2, C3 flags are loaded.

### Floating-Point Exceptions

None; however, this operation might unmask an existing exception that has been detected but not generated, because it was masked. Here, the exception is generated at the completion of the instruction.

### Protected Mode Exceptions

#GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## 45.8 FSAVE/FNSAVE—Store FPU State

Opcode	Instruction	Description
9B DD /6	FSAVE <i>m94/108byte</i>	Store FPU state to <i>m94byte</i> or <i>m108byte</i> after checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.
DD /6	FNSAVE* <i>m94/108byte</i>	Store FPU environment to <i>m94byte</i> or <i>m108byte</i> without checking for pending unmasked floating-point exceptions. Then re-initialize the FPU.

NOTE:

\* See “Intel Architecture Compatibility” below.

### Description

Stores the current FPU state (operating environment and register stack) at the specified destination in memory, and then re-initializes the FPU. The FSAVE instruction checks for and handles pending unmasked floating-point exceptions before storing the FPU state; the FNSAVE instruction does not.

The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures “Protected Mode FPU State Image in Memory, 32-Bit Format” through “Real Mode FPU State Image in Memory, 16-Bit Format”, show

the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used. The contents of the FPU register stack are stored in the 80 bytes immediately follow the operating environment image.

The saved image reflects the state of the FPU after all floating-point instructions preceding the FSAVE/FNSAVE instruction in the instruction stream have been executed.

After the FPU state has been saved, the FPU is reset to the same default values it is set to with the FINIT/FNINIT instructions (see “FINIT/FNINIT—Initialize Floating-Point Unit” in this chapter).

The FSAVE/FNSAVE instructions are typically used when the operating system needs to perform a context switch, an exception handler needs to use the FPU, or an application program needs to pass a “clean” FPU to a procedure.

## Intel Architecture Compatibility

For Intel math coprocessors and FPUs prior to the Intel Pentium processor, an FWAIT instruction should be executed before attempting to read from the memory image stored with a prior FSAVE/FNSAVE instruction. This FWAIT instruction helps insure that the storage operation has been completed.

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSAVE instruction to be interrupted prior to being executed to handle a pending FPU exception. See “No-Wait FPU Instructions Can Get FPU Interrupt in Window”, for a description of these circumstances. An FNSAVE instruction cannot be interrupted in this way on a Pentium Pro processor.

## Operation

```
(* Save FPU State and Registers *)
DEST(FPUControlWord) ← FPUControlWord;
DEST(FPUStatusWord) ← FPUStatusWord;
DEST(FPUTagWord) ← FPUTagWord;
DEST(FPUDDataPointer) ← FPUDDataPointer;
DEST(FPUIInstructionPointer) ← FPUIInstructionPointer;
DEST(FPULastInstructionOpcode) ← FPULastInstructionOpcode;
DEST(ST(0)) ← ST(0);
DEST(ST(1)) ← ST(1);
DEST(ST(2)) ← ST(2);
DEST(ST(3)) ← ST(3);
DEST(ST(4)) ← ST(4);
DEST(ST(5)) ← ST(5);
DEST(ST(6)) ← ST(6);
DEST(ST(7)) ← ST(7);
(* Initialize FPU *)
FPUControlWord ← 037FH;
FPUStatusWord ← 0;
FPUTagWord ← FFFFH;
FPUDDataPointer ← 0;
FPUIInstructionPointer ← 0;
FPULastInstructionOpcode ← 0;
```

## FPU Flags Affected

The C0, C1, C2, and C3 flags are saved and then cleared.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

#GP(0)	<p>If destination is located in a nonwritable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## 45.9 FSCALE—Scale

Opcode	Instruction	Description
D9 FD	FSCALE	Scale ST(0) by ST(1).

### Description

Multiplies the destination operand by 2 to the power of the source operand and stores the result in the destination operand. The destination operand is a real value that is located in register ST(0). The source operand is the nearest integer value that is smaller than the value in the ST(1) register (that is, the value in register ST(1) is truncated toward 0 to its nearest integer value to form the source operand). This instruction provides rapid multiplication or division by integral powers of 2 because it is implemented by simply adding an integer value (the source operand) to the exponent of the value in register ST(0). The following table shows the results obtained when scaling various classes of numbers, assuming that neither overflow nor underflow occurs.

		ST(1)		
		–N	0	+N
ST(0)	–•	–•	–•	–•
	–F	–F	–F	–F
	–0	–0	–0	–0
	+0	+0	+0	+0
	+F	+F	+F	+F
	+∞	+∞	+∞	+∞
	NaN	NaN	NaN	NaN

**NOTES:**

1. FMeans finite-real number.
2. NMeans integer.

In most cases, only the exponent is changed and the mantissa (significand) remains unchanged. However, when the value being scaled in ST(0) is a denormal value, the mantissa is also changed and the result may turn out to be a normalized number. Similarly, if overflow or underflow results from a scale operation, the resulting mantissa will differ from the source's mantissa.

The FSCALE instruction can also be used to reverse the action of the FXTRACT instruction, as shown in the following example:

```
FXTRACT;
FSCALE;
FSTP ST(1);
```

In this example, the FXTRACT instruction extracts the significand and exponent from the value in ST(0) and stores them in ST(0) and ST(1) respectively. The FSCALE then scales the significand in ST(0) by the exponent in ST(1), recreating the original value before the FXTRACT operation was performed. The FSTP ST(1) instruction overwrites the exponent (extracted by the FXTRACT instruction) with the recreated value, which returns the stack to its original state with only one register [ST(0)] occupied.

## Operation

$$ST(0) \leftarrow ST(0) * 2^{ST(1)}$$

## FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
- Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
- C0, C2, C3 Undefined.

## Floating-Point Exceptions

- #IS Stack underflow occurred.
- #IA Source operand is an SNaN value or unsupported format.
- #D Source operand is a denormal value.
- #U Result is too small for destination format.

- #O Result is too large for destination format.
- #P Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

- #NM EM or TS in CR0 is set.

### Real-Address Mode Exceptions

- #NM EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

- #NM EM or TS in CR0 is set.

## 45.10 FSIN—Sine

Opcode	Instruction	Description
D9 FE	FSIN	Replace ST(0) with its sine.

### Description

Calculates the sine of the source operand in register ST(0) and stores the result in ST(0). The source operand must be given in radians and must be within the range  $-2^{63}$  to  $+2^{63}$ . The following table shows the results obtained when taking the sine of various classes of numbers, assuming that underflow does not occur.

SRC (ST(0))	DEST (ST(0))
•	*
–F	–1 to +1
–0	–0
+0	+0
+F	–1 to +1
$+\infty$	*
NaN	NaN

#### NOTES:

1. FMeans finite-real number.
2. \*Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range  $-2^{63}$  to  $+2^{63}$  can be reduced to the range of the instruction by subtracting an appropriate integer multiple of  $2\pi$  or by using the FPREM instruction with a divisor of  $2\pi$ . See “Pi”, for a discussion of the proper value to use for  $\pi$  in performing such reductions.

**Operation**

```

IF ST(0) < 263
THEN
    C2 ← 0;
    ST(0) ← sin(ST(0));
ELSE (* source operand out of range *)
    C2 ← 1;
FI:

```

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C2	Set to 1 if source operand is outside the range $-2^{63}$ to $+2^{63}$ ; otherwise, cleared to 0.
C0, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value, $\infty$ , or unsupported format.
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

**Real-Address Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

**Virtual-8086 Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

**45.11 FSINCOS—Sine and Cosine**

Opcode	Instruction	Description
D9 FB	FSINCOS	Compute the sine and cosine of ST(0); replace ST(0) with the sine, and push the cosine onto the register stack.

**Description**

Computes both the sine and the cosine of the source operand in register ST(0), stores the sine in ST(0), and pushes the cosine onto the top of the FPU register stack. (This instruction is faster than executing the FSIN and FCOS instructions in succession.)

The source operand must be given in radians and must be within the range  $-2^{63}$  to  $+2^{63}$ . The following table shows the results obtained when taking the sine and cosine of various classes of numbers, assuming that underflow does not occur.

SRC	DEST	
ST(0)	ST(1) Cosine	ST(0) Sine
-.*	*	*
-F	-1 to +1	-1 to +1
-0	+1	-0
+0	+1	+0
+F	-1 to +1	-1 to +1
$+\infty$	*	*
NaN	NaN	NaN

**NOTES:**

1. FMeans finite-real number.
2. \*Indicates floating-point invalid-arithmetic-operand (#IA) exception.

If the source operand is outside the acceptable range, the C2 flag in the FPU status word is set, and the value in register ST(0) remains unchanged. The instruction does not raise an exception when the source operand is out of range. It is up to the program to check the C2 flag for out-of-range conditions. Source values outside the range  $-2^{63}$  to  $+2^{63}$  can be reduced to the range of the instruction by subtracting an appropriate integer multiple of  $2\pi$  or by using the FPREM instruction with a divisor of  $2\pi$ . See “Pi”, for a discussion of the proper value to use for  $\pi$  in performing such reductions.

## Operation

```

IF ST(0) <  $2^{63}$ 
THEN
    C2 ← 0;
    TEMP ← cosine(ST(0));
    ST(0) ← sine(ST(0));
    TOP ← TOP - 1;
    ST(0) ← TEMP;
ELSE (* source operand out of range *)
    C2 ← 1;
FI:

```

## FPU Flags Affected

- C1 Set to 0 if stack underflow occurred; set to 1 of stack overflow occurs.
- Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
- C2 Set to 1 if source operand is outside the range  $-2^{63}$  to  $+2^{63}$ ; otherwise, cleared to 0.
- C0, C3 Undefined.

## Floating-Point Exceptions

- #IS Stack underflow occurred.
- #IA Source operand is an SNaN value,  $\infty$ , or unsupported format.



- #D Source operand is a denormal value.
- #U Result is too small for destination format.
- #P Value cannot be represented exactly in destination format.

Protected Mode Exceptions

- #NM EM or TS in CR0 is set.

Real-Address Mode Exceptions

- #NM EM or TS in CR0 is set.

Virtual-8086 Mode Exceptions

- #NM EM or TS in CR0 is set.

45.12 FSQRT—Square Root

Opcode	Instruction	Description
D9 FA	FSQRT	Calculates square root of ST(0) and stores the result in ST(0)

Description

Calculates the square root of the source value in the ST(0) register and stores the result in ST(0).

The following table shows the results obtained when taking the square root of various classes of numbers, assuming that neither overflow nor underflow occurs.

SRC (ST(0))	DEST (ST(0))
-∞	*
-F	*
-0	-0
+0	+0
+F	+F
+∞	+∞
NaN	NaN

- NOTES:
- 1. FMeans finite-real number.
  - 2. \*Indicates floating-point invalid-arithmetic-operand (#IA) exception.

Operation

ST(0) ← SquareRoot(ST(0));

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred.  Indicates rounding direction if inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

### Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.  Source operand is a negative value (except for –0).
#D	Source operand is a denormal value.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

### Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## 45.13 FST/FSTP—Store Real

Opcode	Instruction	Description
D9 /2	FST <i>m32real</i>	Copy ST(0) to <i>m32real</i>
DD /2	FST <i>m64real</i>	Copy ST(0) to <i>m64real</i>
DD D0+i	FST ST(i)	Copy ST(0) to ST(i)
D9 /3	FSTP <i>m32real</i>	Copy ST(0) to <i>m32real</i> and pop register stack
DD /3	FSTP <i>m64real</i>	Copy ST(0) to <i>m64real</i> and pop register stack
DB /7	FSTP <i>m80real</i>	Copy ST(0) to <i>m80real</i> and pop register stack
DD D8+i	FSTP ST(i)	Copy ST(0) to ST(i) and pop register stack

### Description

The FST instruction copies the value in the ST(0) register to the destination operand, which can be a memory location or another register in the FPU register stack. When storing the value in memory, the value is converted to single- or double-real format.

The FSTP instruction performs the same operation as the FST instruction and then pops the register stack. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The FSTP instruction can also store values in memory in extended-real format.

If the destination operand is a memory location, the operand specifies the address where the first byte of the destination value is to be stored. If the destination operand is a register, the operand specifies a register in the register stack relative to the top of the stack.

If the destination size is single- or double-real, the significand of the value being stored is rounded to the width of the destination (according to rounding mode specified by the RC field of the FPU control word), and the exponent is converted to the width and bias of the destination format. If the value being stored is too large for the destination format, a numeric overflow exception (#O) is generated and, if the exception is unmasked, no value is stored in the destination operand. If the value being stored is a denormal value, the denormal exception (#D) is not generated. This condition is simply signaled as a numeric underflow exception (#U) condition.

If the value being stored is  $\pm 0$ ,  $\pm\infty$ , or a NaN, the least-significant bits of the significand and the exponent are truncated to fit the destination format. This operation preserves the value's identity as a 0,  $\infty$ , or NaN.

If the destination operand is a non-empty register, the invalid-operation exception is not generated.

## Operation

```
DEST ← ST(0);
IF instruction = FSTP
    THEN
        PopRegisterStack;
FI;
```

## FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction of if the floating-point inexact exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

## Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#U	Result is too small for the destination format.
#O	Result is too large for the destination format.
#P	Value cannot be represented exactly in destination format.

## Protected Mode Exceptions

#GP(0)	If the destination is located in a nonwritable segment.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## 45.14 FSTCW/FNSTCW—Store Control Word

Opcode	Instruction	Description
9B D9 /7	FSTCW <i>m2byte</i>	Store FPU control word to <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
D9 /7	FNSTCW* <i>m2byte</i>	Store FPU control word to <i>m2byte</i> without checking for pending unmasked floating-point exceptions.

**NOTE:** \* See “Intel Architecture Compatibility” below.

### Description

Stores the current value of the FPU control word at the specified destination in memory. The FSTCW instruction checks for and handles pending unmasked floating-point exceptions before storing the control word; the FNSTCW instruction does not.

## Intel Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTCW instruction to be interrupted prior to being executed to handle a pending FPU exception. See “No-Wait FPU Instructions Can Get FPU Interrupt in Window”, for a description of these circumstances. An FNSTCW instruction cannot be interrupted in this way on a Pentium Pro processor.

## Operation

DEST ← FPUControlWord;

## FPU Flags Affected

The C0, C1, C2, and C3 flags are undefined.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

#GP(0)	<p>If the destination is located in a nonwritable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## 45.15 FSTENV/FNSTENV—Store FPU Environment

Opcode	Instruction	Description
9B D9 /6	FSTENV <i>m14/28byte</i>	Store FPU environment to <i>m14byte</i> or <i>m28byte</i> after checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.
D9 /6	FNSTENV* <i>m14/28byte</i>	Store FPU environment to <i>m14byte</i> or <i>m28byte</i> without checking for pending unmasked floating-point exceptions. Then mask all floating-point exceptions.

**NOTE:** \* See “Intel Architecture Compatibility” below.

### Description

Saves the current FPU operating environment at the memory location specified with the destination operand, and then masks all floating-point exceptions. The FPU operating environment consists of the FPU control word, status word, tag word, instruction pointer, data pointer, and last opcode. Figures “Protected Mode FPU State Image in Memory, 32-Bit Format” through “Real Mode FPU State Image in Memory, 16-Bit Format”, show the layout in memory of the stored environment, depending on the operating mode of the processor (protected or real) and the current operand-size attribute (16-bit or 32-bit). In virtual-8086 mode, the real mode layouts are used.

The FSTENV instruction checks for and handles any pending unmasked floating-point exceptions before storing the FPU environment; the FNSTENV instruction does not. The saved image reflects the state of the FPU after all floating-point instructions preceding the FSTENV/FNSTENV instruction in the instruction stream have been executed.

These instructions are often used by exception handlers because they provide access to the FPU instruction and data pointers. The environment is typically saved in the stack. Masking all exceptions after saving the environment prevents floating-point exceptions from interrupting the exception handler.

### Intel Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTENV instruction to be interrupted prior to being executed to handle a pending FPU exception. See “No-Wait FPU Instructions Can Get FPU Interrupt in Window”, for a description of these circumstances. An FNSTENV instruction cannot be interrupted in this way on a Pentium Pro processor.

### Operation

```
DEST(FPUControlWord) ← FPUControlWord;
DEST(FPUStatusWord) ← FPUStatusWord;
DEST(FPUTagWord) ← FPUTagWord;
DEST(FPUDataPointer) ← FPUDataPointer;
DEST(FPUInstructionPointer) ← FPUInstructionPointer;
DEST(FPULastInstructionOpcode) ← FPULastInstructionOpcode;
```

### FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

### Floating-Point Exceptions

None.

## Protected Mode Exceptions

#GP(0)	<p>If the destination is located in a nonwritable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## 45.16 FSTSW/FNSTSW—Store Status Word

Opcode	Instruction	Description
9B DD /7	FSTSW <i>m2byte</i>	Store FPU status word at <i>m2byte</i> after checking for pending unmasked floating-point exceptions.
9B DF E0	FSTSW AX	Store FPU status word in AX register after checking for pending unmasked floating-point exceptions.
DD /7	FNSTSW* <i>m2byte</i>	Store FPU status word at <i>m2byte</i> without checking for pending unmasked floating-point exceptions.
DF E0	FNSTSW* AX	Store FPU status word in AX register without checking for pending unmasked floating-point exceptions.

**NOTE:** \* See “Intel Architecture Compatibility” below.

## Description

Stores the current value of the FPU status word in the destination location. The destination operand can be either a two-byte memory location or the AX register. The FSTSW instruction checks for and handles pending unmasked floating-point exceptions before storing the status word; the FNSTSW instruction does not.

The FNSTSW AX form of the instruction is used primarily in conditional branching (for instance, after an FPU comparison instruction or an FPREM, FPREM1, or FXAM instruction), where the direction of the branch depends on the state of the FPU condition code flags. (See “Branching and Conditional Moves on FPU Condition Codes” This instruction can also be used to invoke exception handlers (by examining the exception flags) in environments that do not use interrupts. When the FNSTSW AX instruction is executed, the AX register is updated before the processor executes any further instructions. The status stored in the AX register is thus guaranteed to be from the completion of the prior FPU instruction.

## Intel Architecture Compatibility

When operating a Pentium or Intel486 processor in MS-DOS compatibility mode, it is possible (under unusual circumstances) for an FNSTSW instruction to be interrupted prior to being executed to handle a pending FPU exception. See “No-Wait FPU Instructions Can Get FPU Interrupt in Window”, for a description of these circumstances. An FNSTSW instruction cannot be interrupted in this way on a Pentium Pro processor.

## Operation

$DEST \leftarrow FPUStatusWord;$

## FPU Flags Affected

The C0, C1, C2, and C3 are undefined.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

#GP(0)	<p>If the destination is located in a nonwritable segment.</p> <p>If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.</p> <p>If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.</p>
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

## Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

## Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## 45.17 FSUB/FSUBP/FISUB—Subtract

Opcode	Instruction	Description
D8 /4	FSUB <i>m32real</i>	Subtract <i>m32real</i> from ST(0) and store result in ST(0)
DC /4	FSUB <i>m64real</i>	Subtract <i>m64real</i> from ST(0) and store result in ST(0)
D8 E0+i	FSUB ST(i), ST(i)	Subtract ST(i) from ST(0) and store result in ST(0)
DC E8+i	FSUB ST(i), ST(0)	Subtract ST(0) from ST(i) and store result in ST(i)
DE E8+i	FSUBP ST(i), ST(0)	Subtract ST(0) from ST(i), store result in ST(i), and pop register stack
DE E9	FSUBP	Subtract ST(0) from ST(1), store result in ST(1), and pop register stack
DA /4	FISUB <i>m32int</i>	Subtract <i>m32int</i> from ST(0) and store result in ST(0)
DE /4	FISUB <i>m16int</i>	Subtract <i>m16int</i> from ST(0) and store result in ST(0)

### Description

Subtracts the source operand from the destination operand and stores the difference in the destination location. The destination operand is always an FPU data register; the source operand can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

The no-operand version of the instruction subtracts the contents of the ST(0) register from the ST(1) register and stores the result in ST(1). The one-operand version subtracts the contents of a memory location (either a real or an integer value) from the contents of the ST(0) register and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(0) register from the ST(i) register or vice versa.

The FSUBP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUB rather than FSUBP.

The FISUB instructions convert an integer source operand to extended-real format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the SRC value is subtracted from the DEST value ( $\text{DEST} - \text{SRC} = \text{result}$ ).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward  $-\infty$  mode, in which case the result is  $-0$ . This instruction also guarantees that  $+0 - (-0) = +0$ , and that  $-0 - (+0) = -0$ . When the source operand is an integer 0, it is treated as a +0.

When one operand is  $\infty$ , the result is  $\infty$  of the expected sign. If both operands are  $\infty$  of the same sign, an invalid-operation exception is generated.

		SRC						
DEST		-•	-F or -I	-0	+0	+F or +I	+∞	NaN
	-•	*	-•	-•	-•	-•	-•	NaN
	-F	+∞	±F or ±0	DEST	DEST	-F	-•	NaN
	-0	+∞	-SRC	±0	-0	-SRC	-•	NaN
	+0	+∞	-SRC	+0	±0	-SRC	-•	NaN
	+F	+∞	+F	DEST	DEST	±F or ±0	-•	NaN
	+∞	+∞	+∞	+∞	+∞	+∞	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

#### NOTES:

1. FMeans finite-real number.
2. IMeans integer.
3. \*Indicates floating-point invalid-arithmetic-operand (#IA) exception.

## Operation

```

IF instruction is FISUB
    THEN
        DEST ← DEST - ConvertExtendedReal(SRC);
    ELSE (* source operand is real number *)
        DEST ← DEST - SRC;
FI;
IF instruction is FSUBP
    THEN
        PopRegisterStack
FI;
```

## FPU Flags Affected

- C1 Set to 0 if stack underflow occurred.
- Indicates rounding direction if the inexact-result exception (#P) fault is generated: 0 = not roundup; 1 = roundup.
- C0, C2, C3 Undefined.

## Floating-Point Exceptions

- #IS Stack underflow occurred.
- #IA Operand is an SNaN value or unsupported format.

	Operands are infinities of like sign.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

### Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.  If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## 45.18 FSUBR/FSUBRP/FISUBR—Reverse Subtract

Opcode	Instruction	Description
D8 /5	FSUBR <i>m32real</i>	Subtract ST(0) from <i>m32real</i> and store result in ST(0)
DC /5	FSUBR <i>m64real</i>	Subtract ST(0) from <i>m64real</i> and store result in ST(0)
D8 E8+i	FSUBR ST(0), ST(i)	Subtract ST(0) from ST(i) and store result in ST(0)
DC E0+i	FSUBR ST(i), ST(0)	Subtract ST(i) from ST(0) and store result in ST(i)
DE E0+i	FSUBRP ST(i), ST(0)	Subtract ST(i) from ST(0), store result in ST(i), and pop register stack
DE E1	FSUBRP	Subtract ST(1) from ST(0), store result in ST(1), and pop register stack
DA /5	FISUBR <i>m32int</i>	Subtract ST(0) from <i>m32int</i> and store result in ST(0)
DE /5	FISUBR <i>m16int</i>	Subtract ST(0) from <i>m16int</i> and store result in ST(0)

### Description

Subtracts the destination operand from the source operand and stores the difference in the destination location. The destination operand is always an FPU register; the source operand can be a register or a memory location. Source operands in memory can be in single-real, double-real, word-integer, or short-integer formats.

These instructions perform the reverse operations of the FSUB, FSUBP, and FISUB instructions. They are provided to support more efficient coding.

The no-operand version of the instruction subtracts the contents of the ST(1) register from the ST(0) register and stores the result in ST(1). The one-operand version subtracts the contents of the ST(0) register from the contents of a memory location (either a real or an integer value) and stores the result in ST(0). The two-operand version, subtracts the contents of the ST(i) register from the ST(0) register or vice versa.

The FSUBRP instructions perform the additional operation of popping the FPU register stack following the subtraction. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1. The no-operand version of the floating-point reverse subtract instructions always results in the register stack being popped. In some assemblers, the mnemonic for this instruction is FSUBR rather than FSUBRP.

The FISUBR instructions convert an integer source operand to extended-real format before performing the subtraction.

The following table shows the results obtained when subtracting various classes of numbers from one another, assuming that neither overflow nor underflow occurs. Here, the DEST value is subtracted from the SRC value ( $SRC - DEST = \text{result}$ ).

When the difference between two operands of like sign is 0, the result is +0, except for the round toward  $-\infty$  mode, in which case the result is  $-0$ . This instruction also guarantees that  $+0 - (-0) = +0$ , and that  $-0 - (+0) = -0$ . When the source operand is an integer 0, it is treated as a +0.

When one operand is  $\infty$ , the result is  $\infty$  of the expected sign. If both operands are  $\infty$  of the same sign, an invalid-operation exception is generated.

		SRC						
DEST		-•	-F or -I	-0	+0	+F or +I	+∞	NaN
	-•	*	+∞	+∞	+∞	+∞	+∞	NaN
	-F	-•	±F or ±0	-DEST	-DEST	+F	+∞	NaN
	-0	-•	SRC	±0	+0	SRC	+∞	NaN
	+0	-•	SRC	-0	±0	SRC	+∞	NaN
	+F	-•	-F	-DEST	-DEST	±F or ±0	+∞	NaN
	+∞	-•	-•	-•	-•	-•	*	NaN
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

**NOTES:**

1. FMeans finite-real number.
2. IMeans integer.
3. \*Indicates floating-point invalid-arithmetic-operand (#IA) exception.

**Operation**

```

IF instruction is FISUBR
    THEN
        DEST ← ConvertExtendedReal(SRC) - DEST;
    ELSE (* source operand is real number *)
        DEST ← SRC - DEST;
FI;
IF instruction = FSUBRP
    THEN
        PopRegisterStack
FI;

```

**FPU Flags Affected**

- C1 Set to 0 if stack underflow occurred.
- Indicates rounding direction if the inexact-result exception (#P) fault is generated: 0 = not roundup; 1 = roundup.
- C0, C2, C3 Undefined.

**Floating-Point Exceptions**

- #IS Stack underflow occurred.
- #IA Operand is an SNaN value or unsupported format.
- Operands are infinities of like sign.
- #D Source operand is a denormal value.
- #U Result is too small for destination format.
- #O Result is too large for destination format.
- #P Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

	If the DS, ES, FS, or GS register is used to access memory and it contains a null segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

### Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NM	EM or TS in CR0 is set.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.

## 45.19 FTST—TEST

Opcode	Instruction	Description
D9 E4	FTST	Compare ST(0) with 0.0.

### Description

Compares the value in the ST(0) register with 0.0 and sets the condition code flags C0, C2, and C3 in the FPU status word according to the results (see table below).

Condition	C3	C2	C0
ST(0) > 0.0	0	0	0
ST(0) < 0.0	0	0	1
ST(0) = 0.0	1	0	0
Unordered	1	1	1

This instruction performs an “unordered comparison.” An unordered comparison also checks the class of the numbers being compared (see “FXAM—Examine” in this chapter). If the value in register ST(0) is a NaN or is in an undefined format, the condition flags are set to “unordered” and the invalid operation exception is generated.

The sign of zero is ignored, so that  $-0.0 = +0.0$ .

## Operation

```
CASE (relation of operands) OF
  Not comparable:  C3, C2, C0 ← 111;
  ST(0) > 0.0:    C3, C2, C0 ← 000;
  ST(0) < 0.0:    C3, C2, C0 ← 001;
  ST(0) = 0.0:    C3, C2, C0 ← 100;
ESAC;
```

## FPU Flags Affected

C1                      Set to 0 if stack underflow occurred; otherwise, cleared to 0.  
C0, C2, C3            See above table.

## Floating-Point Exceptions

#IS                    Stack underflow occurred.  
#IA                    The source operand is a NaN value or is in an unsupported format.  
#D                    The source operand is a denormal value.

## Protected Mode Exceptions

#NM                    EM or TS in CR0 is set.

## Real-Address Mode Exceptions

#NM                    EM or TS in CR0 is set.

## Virtual-8086 Mode Exceptions

#NM                    EM or TS in CR0 is set.

# 45.20 FUCOM/FUCOMP/FUCOMPP—Unordered Compare Real

Opcode	Instruction	Description
DD E0+i	FUCOM ST(i)	Compare ST(0) with ST(i)
DD E1	FUCOM	Compare ST(0) with ST(1)
DD E8+i	FUCOMP ST(i)	Compare ST(0) with ST(i) and pop register stack
DD E9	FUCOMP	Compare ST(0) with ST(1) and pop register stack
DA E9	FUCOMPP	Compare ST(0) with ST(1) and pop register stack twice

## Description

Performs an unordered comparison of the contents of register ST(0) and ST(i) and sets condition code flags C0, C2, and C3 in the FPU status word according to the results (see the table below). If no operand is specified, the contents of registers ST(0) and ST(1) are compared. The sign of zero is ignored, so that  $-0.0 = +0.0$ .

Comparison Results	C3	C2	C0
ST0 > ST(i)	0	0	0
ST0 < ST(i)	0	0	1
ST0 = ST(i)	1	0	0
Unordered	1	1	1

**NOTE:** \*Flags not set if unmasked invalid-arithmetic-operand (#IA) exception is generated.

An unordered comparison checks the class of the numbers being compared (see “FXAM—Examine” in this chapter). The FUCOM instructions perform the same operations as the FCOM instructions. The only difference is that the FUCOM instructions raise the invalid-arithmetic-operand exception (#IA) only when either or both operands are an SNaN or are in an unsupported format; QNaNs cause the condition code flags to be set to unordered, but do not cause an exception to be generated. The FCOM instructions raise an invalid-operation exception when either or both of the operands are a NaN value of any kind or are in an unsupported format.

As with the FCOM instructions, if the operation results in an invalid-arithmetic-operand exception being raised, the condition code flags are set only if the exception is masked.

The FUCOMP instruction pops the register stack following the comparison operation and the FUCOMPP instruction pops the register stack twice following the comparison operation. To pop the register stack, the processor marks the ST(0) register as empty and increments the stack pointer (TOP) by 1.

## Operation

```

CASE (relation of operands) OF
    ST > SRC:      C3, C2, C0 ← 000;
    ST < SRC:      C3, C2, C0 ← 001;
    ST = SRC:      C3, C2, C0 ← 100;
ESAC;
IF ST(0) or SRC = QNaN, but not SNaN or unsupported format
    THEN
        C3, C2, C0 ← 111;
    ELSE (* ST(0) or SRC is SNaN or unsupported format *)
        #IA;
        IF FPUControlWord.IM = 1
            THEN
                C3, C2, C0 ← 111;
        FI;
FI;
IF instruction = FUCOMP
    THEN
        PopRegisterStack;
FI;
IF instruction = FUCOMPP
    THEN
        PopRegisterStack;
        PopRegisterStack;
FI;

```

## FPU Flags Affected

C1                      Set to 0 if stack underflow occurred.

C0, C2, C3      See table on previous page.

### Floating-Point Exceptions

#IS      Stack underflow occurred.

#IA      One or both operands are SNaN values or have unsupported formats. Detection of a QNaN value in and of itself does not raise an invalid-operand exception.

#D      One or both operands are denormal values.

### Protected Mode Exceptions

#NM      EM or TS in CR0 is set.

### Real-Address Mode Exceptions

#NM      EM or TS in CR0 is set.

### Virtual-8086 Mode Exceptions

#NM      EM or TS in CR0 is set.

## 45.21 FWAIT—Wait

See entry for WAIT/FWAIT—Wait.

## 45.22 FXAM—Examine

Opcode	Instruction	Description
D9 E5	FXAM	Classify value or number in ST(0)

### Description

Examines the contents of the ST(0) register and sets the condition code flags C0, C2, and C3 in the FPU status word to indicate the class of value or number in the register (see the table below).

Class	C3	C2	C0
Unsupported	0	0	0
NaN	0	0	1
Normal finite number	0	1	0
Infinity	0	1	1
Zero	1	0	0
Empty	1	0	1
Denormal number	1	1	0

The C1 flag is set to the sign of the value in ST(0), regardless of whether the register is empty or full.

## Operation

```

C1 ← sign bit of ST; (* 0 for positive, 1 for negative *)
CASE (class of value or number in ST(0)) OF
  Unsupported: C3, C2, C0 ← 000;
  NaN:        C3, C2, C0 ← 001;
  Normal:     C3, C2, C0 ← 010;
  Infinity:   C3, C2, C0 ← 011;
  Zero:       C3, C2, C0 ← 100;
  Empty:      C3, C2, C0 ← 101;
  Denormal:   C3, C2, C0 ← 110;
ESAC;

```

## FPU Flags Affected

C1                      Sign of value in ST(0).  
C0, C2, C3              See table above.

## Floating-Point Exceptions

None.

## Protected Mode Exceptions

#NM                      EM or TS in CR0 is set.

## Real-Address Mode Exceptions

#NM                      EM or TS in CR0 is set.

## Virtual-8086 Mode Exceptions

#NM                      EM or TS in CR0 is set.

# 45.23 FXCH—Exchange Register Contents

Opcode	Instruction	Description
D9 C8+i	FXCH ST(i)	Exchange the contents of ST(0) and ST(i)
D9 C9	FXCH	Exchange the contents of ST(0) and ST(1)

## Description

Exchanges the contents of registers ST(0) and ST(i). If no source operand is specified, the contents of ST(0) and ST(1) are exchanged.

This instruction provides a simple means of moving values in the FPU register stack to the top of the stack [ST(0)], so that they can be operated on by those floating-point instructions that can only operate on values in ST(0). For example, the following instruction sequence takes the square root of the third register from the top of the register stack:

```

FXCH ST(3);
FSQRT;
FXCH ST(3);

```

## Operation

```

IF number-of-operands is 1
  THEN
    temp ← ST(0);
    ST(0) ← SRC;
    SRC ← temp;
  ELSE
    temp ← ST(0);
    ST(0) ← ST(1);
    ST(1) ← temp;

```

## FPU Flags Affected

C1 Set to 0 if stack underflow occurred; otherwise, cleared to 0.

C0, C2, C3 Undefined.

## Floating-Point Exceptions

#IS Stack underflow occurred.

## Protected Mode Exceptions

#NM EM or TS in CR0 is set.

## Real-Address Mode Exceptions

#NM EM or TS in CR0 is set.

## Virtual-8086 Mode Exceptions

#NM EM or TS in CR0 is set.

# 45.24 FXRSTOR—Restore FP or MMX™ Technology State

Opcode	Instruction	Description
0F AE, /1	FXRSTOR m512byte	Load the FP or MMX™ technology state from m512byte

## Description

The FXRSTOR instruction reloads the FP or MMX™ technology state (environment and registers) from the memory area defined by m512byte. This data should have been written by a previous FXSAVE.

The FP or MMX technology environment and registers consist of the following data structure (little-endian byte order as arranged in memory, with byte offset into row described by right column).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved		CS		IP				FOP		Rsvd	FTW	FSW		FCW		0
Reserved								Reserved		DS		DP				16
Reserved						ST0/MM0										32
Reserved						ST1/MM1										48

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved						ST2/MM2										64
Reserved						ST3/MM3										80
Reserved						ST4/MM4										96
Reserved						ST5/MM5										112
Reserved						ST6/MM6										128
Reserved						ST7/MM7										144
						Reserved										160
						Reserved										176
						Reserved										192
						Reserved										208
						Reserved										224
						Reserved										240
						Reserved										256
						Reserved										272
						Reserved										288
						Reserved										304
						Reserved										320
						Reserved										336
						Reserved										352
						Reserved										368
						Reserved										384
						Reserved										400
						Reserved										416
						Reserved										432
						Reserved										448
						Reserved										464
						Reserved										480
						Reserved										496

Three fields in the floating-point save area contain reserved bits that are not indicated in the table:

- FOP                      The lower 11-bits contain the opcode, upper 5-bits are reserved.
- IP & DP                32-bit mode: 32-bit IP-offset.  
                              16-bit mode: lower 16-bits are IP-offset and upper 16-bits are reserved.

The term, “Reserved,” is as defined in “Notation Conventions”. Reserved bits are undefined, and using them risks incompatibility with future Intel Architecture processors. Furthermore, all “Reserved” fields in the tag word area should be set specifically to zero on a restore, or in cases where the software is attempting to initialize a floating-point context.

Unlike the FRSTOR instruction, FXRSTOR does not fault when loading an image from memory that contains a pending exception in the Floating-Point Status Word (FSW); only the next occurrence of this unmasked exception will result in the error condition being asserted. It also does not flush pending x87-FP exceptions. To check and raise exceptions when loading a new operating environment, use FWAIT after FXRSTOR.

## Operation

```

FPUCtrlWord <= SRC(FPUCtrlWord);
FPUSW <= SRC(FPUSW);
FPUTagWord <= SRC(FPUTagWord);
FPUDP <= SRC(FPUDP);
FPUInstPnt <= SRC(FPUInstPnt);
FPUInstOp <= SRC(FPUInstOp);
ST(0) <= SRC(ST(0));
ST(1) <= SRC(ST(1));
ST(2) <= SRC(ST(2));
ST(3) <= SRC(ST(3));
ST(4) <= SRC(ST(4));
ST(5) <= SRC(ST(5));
ST(6) <= SRC(ST(6));
ST(7) <= SRC(ST(7));

```

## Exceptions

#GP(0)	If m512byte is not aligned on a 16-byte boundary.
#AC(0)	If alignment check is enabled (CR0.AM = 1, EFLAGS.AC = 1 and CPL = 3) and m512byte is not aligned on a 16 byte boundary.
#UD	If instruction is preceded by a lock prefix.

## Numeric Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	If a page fault occurs.
#NM	If CR0.EM = 1 or CR0.TS = 1.
#AC	If alignment check is enabled, and an unaligned memory reference is made while the current privilege level is 3.

## Protected-Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF (fault-code)	If a page fault occurs.
#NM	If CR0.EM = 1 or CR0.TS = 1.
#AC	If alignment check is enabled, and an unaligned memory reference is made while the current privilege level is 3.

## Real-Address Mode Exceptions

Interrupt 13	If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
--------------	---

#NM If CR0.EM = 1 or CR0.TS = 1.

## Virtual-8086 Mode Exceptions

Same exceptions as in Real-Address Mode.

#PF (fault-code) If a page fault occurs.

#AC If alignment check is enabled, and an unaligned memory reference is made while the current privilege level is 3.

## Notes

State saved with FSAVE and restored with FXRSTOR (and vice versa) results in an incorrect restoration of state in the processor. Software should not depend on the behavior of the FXRSTOR instruction when it is preceded by either the REP, REPNE, or operand size override prefix. The application of these prefixes with FXRSTOR is defined as “reserved,” and processor behavior is model specific. Using these prefixes with FXRSTOR risks incompatibility with future Intel processors. The address size prefix has the usual effect on address calculation, but has no effect on the format of the FXRSTOR image.

The FXRSTOR instruction assumes that the upper byte of the FPU Tag Word is equal to zero. If it is nonzero, the execution of the FXRSTOR instruction will cause an incorrect state to be generated in the processor.

Always ensure that FXRSTOR is used in conjunction with the FXSAVE instruction in a programming environment. Otherwise, ensure that the upper byte of the FPU Tag Word is zero before the FXRSTOR instruction is executed.

If an environment creates a condition where the upper byte of the FPU Tag Word is nonzero before execution of the FXRSTOR instruction, the result is an unpredictable system failure due to the loading of a corrupted state.

## 45.25 FXSAVE—Store FP or MMX™ Technology State

Opcode	Instruction	Description
0F AE, /0	FXSAVE m512byte	Store FP or MMX™ technology state to m512byte

### Description

The FXSAVE instruction writes the current FP or MMX technology state (environment and registers) to the specified destination defined by m512byte. It does this without checking for pending unmasked floating-point exceptions, similar to the operation of FNSAVE. Unlike the FSAVE/FNSAVE instructions, the processor retains the contents of the FP or MMX technology state in the processor after the particular state has been saved. This instruction has been optimized to maximize floating-point save performance.

The FXSAVE instruction is used when an operating system needs to perform a context switch or when an exception handler needs to use the FP and MMX technology units. It cannot be used by an application program to pass a “clean” FP state to a procedure, because it retains the current state. An application must explicitly execute a FINIT instruction after an FXSAVE to provide this functionality.

The save format is as described for the FXRSTOR instruction. All of the fields in bytes 0-160 retain the same internal format as the FSAVE instruction, except for the floating-point tag word (FTW). Unlike FSAVE, the FXSAVE instruction only saves the FTW valid bits rather than the entire x87-FP FTW field. The FTW bits are saved by FXSAVE in a non-TOS relative order, meaning that FR0 is always saved first, followed by FR1, FR2, and so forth.

As an example, if TOS=4 and only ST0, ST1 and ST2 are valid, FSAVE saves the FTW field in the following format:

ST3	ST2	ST1	ST0	ST7	ST6	ST5	ST4 (TOS=4)
FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0
11	xx	xx	xx	11	11	11	11

where xx is one of (00, 01, 10). A (11) indicates an Empty stack element. The values of 00, 01, and 10 indicate Valid, Zero, and Special, respectively. In this example, FXSAVE would save the following vector:

FR7	FR6	FR5	FR4	FR3	FR2	FR1	FR0
0	1	1	1	0	0	0	0

The FSAVE format for FTW can be recreated from the FTW valid bits and the stored 80-bit FP data (assuming the stored data was not the contents of MMX registers) using the following table.

Exponent all 1s	Exponent all 0s	Fraction all 0s	J and M bits	FTW valid bit	x87 FTW	
0	0	0	0x	1	Special	10
0	0	0	1x	1	Valid	00
0	0	1	00	1	Special	10
0	0	1	10	1	Valid	00
0	1	0	0x	1	Special	10
0	1	0	1x	1	Special	10
0	1	1	00	1	Zero	01
0	1	1	10	1	Special	10
1	0	0	1x	1	Special	10
1	0	0	1x	1	Special	10
1	0	1	00	1	Special	10
1	0	1	10	1	Special	10
For all legal combinations above				0	Empty	11

In binary floating-point format, a real number has three parts: a sign bit, a significand, and an exponent. The significand has two parts: a 1-bit binary integer (referred to as the J-bit) and a binary fraction.

The J-bit is defined to be the 1-bit binary integer to the left of the decimal place in the significand. The M-bit is defined to be the most significant bit of the fractional portion of the significand (i.e., the bit immediately to the right of the decimal place).

If the FXSAVE instruction is immediately preceded by an FP instruction which does not use a memory operand, then the FXSAVE instruction does not write/update the DP field, in the FXSAVE image.

The destination m512byte is assumed to be aligned on a 16-byte boundary. If m512byte is not aligned on a 16-byte boundary, FXSAVE generates a general protection exception.

## Operation

```
(* Save FPU State and Registers *)
DEST(FPUControlWord) <= FPUControlWord;
DEST(FPUStatusWord) <= FPUStatusWord;
DEST(FPUTagWord) <= Function of (FPUTagWord);
DEST(FPUDataPointer) <= FPUDataPointer;
DEST(FPUInstructionPointer) <= FPUInstructionPointer;
DEST(FPULastInstructionOpcode) <= FPULastInstructionOpcode;
DEST(ST(0)) <= ST(0);
DEST(ST(1)) <= ST(1);
DEST(ST(2)) <= ST(2);
DEST(ST(3)) <= ST(3);
DEST(ST(4)) <= ST(4);
DEST(ST(5)) <= ST(5);
DEST(ST(6)) <= ST(6);
DEST(ST(7)) <= ST(7);
(* Does not initialize FPU -- Retains contents from above *)
```

## Exceptions

- #GP(0) If m512byte is not aligned on a 16-byte boundary.
- #AC(0) If alignment check is enabled (CR0.AM = 1, EFLAGS.AC = 1 and CPL = 3) and m512byte is not aligned on a 16 byte boundary.
- #UD If instruction is preceded by a lock prefix.

## Numeric Exceptions

None.

## Protected-Mode Exceptions

- #GP(0) If a memory operand effective address is outside the CS, DS, ES, FS or GS segment limit.
- #SS(0) If a memory operand effective address is outside the SS segment limit.
- #PF (fault-code) If a page fault occurs.
- #NM If CR0.EM = 1 or CR0.TS = 1.
- #AC If alignment check is enabled, and an unaligned memory reference is made while the current privilege level is 3.

## Real-Address Mode Exceptions

- Interrupt 13 If any part of the operand would lie outside of the effective address space from 0 to 0FFFFH.
- #NM If CR0.EM = 1 or CR0.TS = 1.

## Virtual-8086 Exceptions

Same exceptions as in Real-Address Mode

- #PF (fault-code) If a page fault occurs.
- #AC If alignment check is enabled, and an unaligned memory reference is made while the current privilege level is 3.

## Notes

State saved with FXSAVE and restored with FRSTOR (and vice versa) results in an incorrect restoration of state in the processor. Software should not depend on the behavior of the FXRSTOR instruction when it is preceded by either the REP, REPNE, or operand size override prefix. The application of these prefixes with FXRSTOR is defined as “reserved,” and processor behavior is model specific. Using these prefixes with FXRSTOR risks incompatibility with future Intel processors. The address size prefix has the usual effect on address calculation, but has no effect on the format of the FXSAVE image.

If there is a pending unmasked FP exception at the time FXSAVE is executed, the sequence of FXSAVE-FWAIT-FXRSTOR results in an incorrect state in the processor. The FWAIT instruction causes the processor to check and handle pending unmasked FP exceptions. Since the processor does not clear the FPU state with FXSAVE, the exception is handled, but that fact is not reflected in the saved image. When the image is reloaded using FXRSTOR, the exception bits in the FSW get loaded incorrectly.

## 45.26 FXTRACT—Extract Exponent and Significand

Opcode	Instruction	Description
D9 F4	FXTRACT	Separate value in ST(0) into exponent and significand, store exponent in ST(0), and push the significand onto the register stack.

### Description

Separates the source value in the ST(0) register into its exponent and significand, stores the exponent in ST(0), and pushes the significand onto the register stack. Following this operation, the new top-of-stack register ST(0) contains the value of the original significand expressed as a real number. The sign and significand of this value are the same as those found in the source operand, and the exponent is 3FFFH (biased value for a true exponent of zero). The ST(1) register contains the value of the original operand’s true (unbiased) exponent expressed as a real number. (The operation performed by this instruction is a superset of the IEEE-recommended  $\log_b(x)$  function.)

This instruction and the F2XM1 instruction are useful for performing power and range scaling operations. The FXTRACT instruction is also useful for converting numbers in extended-real format to decimal representations (e.g., for printing or displaying).

If the floating-point zero-divide exception (#Z) is masked and the source operand is zero, an exponent value of  $-\infty$  is stored in register ST(1) and 0 with the sign of the source operand is stored in register ST(0).

### Operation

```
TEMP ← Significand(ST(0));
ST(0) ← Exponent(ST(0));
TOP ← TOP - 1;
ST(0) ← TEMP;
```

### FPU Flags Affected

C1	Set to 0 if stack underflow occurred; set to 1 if stack overflow occurred.
C0, C2, C3	Undefined.

## Floating-Point Exceptions

#IS	Stack underflow occurred.
	Stack overflow occurred.
#IA	Source operand is an SNaN value or unsupported format.
#Z	ST(0) operand is $\pm 0$ .
#D	Source operand is a denormal value.

## Protected Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

# 45.27 FYL2X—Compute $y * \log_2 x$

Opcode	Instruction	Description
D9 F1	FYL2X	Replace ST(1) with $(ST(1) * \log_2 ST(0))$ and pop the register stack

## Description

Calculates  $(ST(1) * \log_2 (ST(0)))$ , stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be a non-zero positive number.

The following table shows the results obtained when taking the log of various classes of numbers, assuming that neither overflow nor underflow occurs.

		ST(0)						
		-•	-F	$\pm 0$	$+0 < +F < +1$	+1	$+F > +1$	$+\infty$
ST(1)	-•	*	*	$+\infty$	$+\infty$	*	-•	-•
	-F	*	*	**	+F	-0	-F	-•
	-0	*	*	*	+0	-0	-0	*
	+0	*	*	*	-0	+0	+0	*
	+F	*	*	**	-F	+0	+F	$+\infty$
	$+\infty$	*	*	-•	-•	*	+•	$+\infty$
	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

### NOTES:

1. FMeans finite-real number.

2. \*Indicates floating-point invalid-operation (#IA) exception.
3. \*\*Indicates floating-point zero-divide (#Z) exception.

If the divide-by-zero exception is masked and register ST(0) contains  $\pm 0$ , the instruction returns  $\infty$  with a sign that is the opposite of the sign of the source operand in register ST(1).

The FYL2X instruction is designed with a built-in multiplication to optimize the calculation of logarithms with an arbitrary positive base (b):

$$\log_b x = (\log_2 b)^{-1} * \log_2 x$$

## Operation

```
ST(1) ← ST(1) * log2ST(0);
PopRegisterStack;
```

## FPU Flags Affected

C1	Set to 0 if stack underflow occurred.
	Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

## Floating-Point Exceptions

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN or unsupported format. Source operand in register ST(0) is a negative finite value (not -0).
#Z	Source operand in register ST(0) is $\pm 0$ .
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

## Protected Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## Real-Address Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## Virtual-8086 Mode Exceptions

#NM	EM or TS in CR0 is set.
-----	-------------------------

## 45.28 FYL2XP1—Compute $y * \log_2(x + 1)$

Opcode	Instruction	Description
D9 F9	FYL2XP1	Replace ST(1) with $ST(1) * \log_2(ST(0) + 1.0)$ and pop the register stack

### Description

Calculates the log epsilon ( $ST(1) * \log_2(ST(0) + 1.0)$ ), stores the result in register ST(1), and pops the FPU register stack. The source operand in ST(0) must be in the range:

$$-(1 - \sqrt{2}/2) \text{ to } (1 - \sqrt{2}/2)$$

The source operand in ST(1) can range from  $-\infty$  to  $+\infty$ . If the ST(0) operand is outside of its acceptable range, the result is undefined and software should not rely on an exception being generated. Under some circumstances exceptions may be generated when ST(0) is out of range, but this behavior is implementation specific and not guaranteed.

The following table shows the results obtained when taking the log epsilon of various classes of numbers, assuming that underflow does not occur.

		ST(0)				
		$-(1 - (\sqrt{2}/2))$ to $-0$	$-0$	$+0$	$+0$ to $+(1 - (\sqrt{2}/2))$	NaN
ST(1)	$-\bullet$	$+\infty$	*	*	$-\bullet$	NaN
	$-F$	$+F$	$+0$	$-0$	$-F$	NaN
	$-0$	$+0$	$+0$	$-0$	$-0$	NaN
	$+0$	$-0$	$-0$	$+0$	$+0$	NaN
	$+F$	$-F$	$-0$	$+0$	$+F$	NaN
	$+\infty$	$-\bullet$	*	*	$+\infty$	NaN
	NaN	NaN	NaN	NaN	NaN	NaN

#### NOTES:

1. FMeans finite-real number.
2. \*Indicates floating-point invalid-operation (#IA) exception.

This instruction provides optimal accuracy for values of epsilon [the value in register ST(0)] that are close to 0. When the epsilon value ( $\epsilon$ ) is small, more significant digits can be retained by using the FYL2XP1 instruction than by using  $(\epsilon+1)$  as an argument to the FYL2X instruction. The  $(\epsilon+1)$  expression is commonly found in compound interest and annuity calculations. The result can be simply converted into a value in another logarithm base by including a scale factor in the ST(1) source operand. The following equation is used to calculate the scale factor for a particular logarithm base, where  $n$  is the logarithm base desired for the result of the FYL2XP1 instruction:

$$\text{scale factor} = \log_n 2$$

### Operation

```
ST(1) ← ST(1) * log2(ST(0) + 1.0);
PopRegisterStack;
```

**FPU Flags Affected**

C1	Set to 0 if stack underflow occurred.  Indicates rounding direction if the inexact-result exception (#P) is generated: 0 = not roundup; 1 = roundup.
C0, C2, C3	Undefined.

**Floating-Point Exceptions**

#IS	Stack underflow occurred.
#IA	Either operand is an SNaN value or unsupported format.
#D	Source operand is a denormal value.
#U	Result is too small for destination format.
#O	Result is too large for destination format.
#P	Value cannot be represented exactly in destination format.

**Protected Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

**Real-Address Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------

**Virtual-8086 Mode Exceptions**

#NM	EM or TS in CR0 is set.
-----	-------------------------